**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Информатика»**

Тема: **Алгоритмы и структуры данных в Python**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9381 |  | Колованов Р.А. |
| Преподаватель |  | Размочаева Н.В. |

Санкт-Петербург

2019

## Цель работы.

Изучение алгоритмов и структур данных в Python, а также использование парадигмы ООП для написания программ.

## Задание.

В данной лабораторной работе Вам предстоит реализовать связный двунаправленный список.

### Node

Класс, который описывает элемент списка. Класс Node должен иметь 3 поля:

* \_\_data # данные, приватное поле
* \_\_prev\_\_ # ссылка на предыдущий элемент списка
* \_\_next\_\_ # ссылка на следующий элемент списка

Вам необходимо реализовать следующие методы в классе Node:

* \_\_init\_\_(self, data, prev, next)

конструктор, у которого значения по умолчанию для аргументов prev и next равны None.

* get\_data(self)

метод возвращает значение поля \_\_data.

* \_\_str\_\_(self)

перегрузка метода \_\_str\_\_. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с Node.

### Linked List

Класс, который описывает связный двунаправленный список. Класс LinkedList должен иметь 3 поля:

* \_\_length # длина списка
* \_\_first\_\_ # данные первого элемента списка
* \_\_last\_\_ # данные последнего элемента списка

Вам необходимо реализовать конструктор:

* \_\_init\_\_(self, first, last)

конструктор, у которого значения по умолчанию для аргументов first и last равны None.

Если значение переменной first равно None, а переменной last не равно None, метод должен вызывать исключение ValueError с сообщением: "invalid value for last".

Если значение переменной first не равно None, а переменной last равна None, метод должен создавать список из одного элемента. В данном случае, first равен last, ссылки prev и next равны None, значение поля \_\_data для элемента списка равно first.

Если значения переменных не равны None, необходимо создать список из двух элементов. В таком случае, значение поля \_\_data для первого элемента списка равно first, значение поля \_\_data для второго элемента списка равно last.

и следующие методы в классе LinkedList:

* \_\_len\_\_(self)

перегрузка метода \_\_len\_\_.

* append(self, element)

добавление элемента в конец списка. Метод должен создать объект класса Node, у которого значение поля \_\_data будет равно element и добавить этот объект в конец списка.

* \_\_str\_\_(self)

перегрузка метода \_\_str\_\_. Описание того, как должен выглядеть результат вызова метода смотрите ниже в примере взаимодействия с LinkedList.

* pop(self)

удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.

* popitem(self, element)

удаление элемента, у которого значение поля \_\_data равно element. Метод должен выбрасывать исключение KeyError, с сообщением "<element> doesn't exist!", если элемента в списке нет.

* clear(self)

очищение списка.

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

В отчете вам требуется:

1. Указать, что такое связный список. Основные отличия связного списка от массива.
2. Указать сложность каждого метода.
3. Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

## Выполнение работы.

В данной лабораторной работе был разработан класс связного списка. *Связный список* - базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным отличием перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Преимущества:

* Эффективное добавление и удаление элементов.
* Размер ограничен только объёмом памяти [компьютера](https://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C) и разрядностью указателей.
* Динамическое добавление и удаление элементов.

Недостатки:

* Сложность прямого доступа к элементу, а именно определения физического адреса по его индексу в списке.
* На поля-указатели расходуется дополнительная память.
* Некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы.
* Соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность [кэширования](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88) данных в процессоре.
* Над связными списками, по сравнению с массивами, гораздо труднее производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания.

Описание и сложность методов приведена в Таблице 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Класс | Метод | Описание | Сложность |
| *Node* | *\_\_init\_\_* | Конструктор. Задает начальные значения полям объекта. | O(1) |
| *get\_data* | Получает данные текущего узла. | O(1) |
| *\_\_str\_\_* | Преобразует объект к типу *str*. | O(1) |
| *LinkedList* | *\_\_init\_\_* | Конструктор. Задает начальные значения полям объекта. | O(1) |
| *\_\_len\_\_* | Возвращает количество элементов в списке. | O(1) |
| *\_\_iter\_\_* | Возвращает итератор типа *LinkedList*. | O(1) |
| *\_\_str\_\_* | Преобразует объект к типу *str*. | O(n) |
| *append* | Добавляет элемент в конец списка. | O(1) |
| *clear* | Очищает список. | O(1) |
| *pop* | Удаляет последний элемент в списке. | O(1) |
| *popitem* | Удаляет первый встретившийся элемент с определенным значением. | O(n) |
| *LinkedListIterator* | *\_\_init\_\_* | Конструктор. Задает начальные значения полям объекта. | O(1) |
| *\_\_next\_\_* | Возвращает текущий элемент и перемещается на следующий. | O(1) |

*Таблица 1.*

Для класса можно переопределить метод *\_\_getitem\_\_*, который бы позволял обращаться к элементам списка по индексу. В таком случае реализация бинарного поиска для данного класса и для списков в Python ничем бы не отличалась.

|  |
| --- |
| **Листинг 1. Метод *\_\_getitem\_\_* и бинарный поиск.**  def \_\_getitem\_\_(self, index):  begin = self.\_\_first\_\_  for i in range(index):  begin = begin.\_\_next\_\_  return begin.get\_data()  def binary\_search(L, x):  i = 0  j = len(L) - 1  m = int(j / 2)  while L[m] != x and i < j:  if x > L[m]:  i = m + 1  else:  j = m - 1  m = int((i + j) / 2)  if i > j:  return -1  else:  return m  Функция binary\_search будет работать как со списками list, так и со списками LinkedList. |

Разработанный программный код см. в приложении А.

## Тестирование.

Тестирование производилось на образовательной платформе Stepik.

## Выводы.

Были изучены основные алгоритмы и структуры данных в Python. На языке Python, руководствуясь парадигмой ООП, была разработана программа, предоставляющая класс связного списка. Для написания программы использовались условные операторы, операторы цикла, классы, а также обработчик исключений *try-except*. Помимо этого, для класса связного списка был написан класс-итератор.

# Приложение А Исходный код программы

Название файла: main.py

class Node:

def \_\_init\_\_(self, data, prev=None, next=None):

self.\_\_data = data

self.\_\_prev\_\_ = prev

self.\_\_next\_\_ = next

def get\_data(self):

return self.\_\_data

def \_\_str\_\_(self):

s1, s2 = "None", "None"

if self.\_\_prev\_\_ != None:

s1 = self.\_\_prev\_\_.get\_data()

if self.\_\_next\_\_ != None:

s2 = self.\_\_next\_\_.get\_data()

return "data: {}, prev: {}, next: {}".format(self.\_\_data, s1, s2)

class LinkedListIterator:

def \_\_init\_\_(self, linked\_list):

self.current = linked\_list.\_\_first\_\_

def \_\_next\_\_(self):

if self.current != None:

node = self.current

self.current = self.current.\_\_next\_\_

return node

else:

raise StopIteration

class LinkedList:

def \_\_init\_\_(self, first=None, last=None):

if first == None:

self.\_\_length = 0

self.\_\_first\_\_ = None

self.\_\_last\_\_ = None

if last != None:

raise ValueError("invalid value for last")

elif first != None and last == None:

self.\_\_length = 1

self.\_\_first\_\_ = Node(first)

self.\_\_last\_\_ = self.\_\_first\_\_

else:

self.\_\_length = 2

self.\_\_first\_\_ = Node(first)

self.\_\_last\_\_ = Node(last, prev=self.\_\_first\_\_)

self.\_\_first\_\_.\_\_next\_\_ = self.\_\_last\_\_

def \_\_len\_\_(self):

return self.\_\_length

def \_\_iter\_\_(self):

return LinkedListIterator(self)

def \_\_str\_\_(self):

if self.\_\_length > 0:

return "LinkedList[length = {}, [{}]]".format(self.\_\_length, "; ".join(list(map(str, self))))

else:

return "LinkedList[]"

def append(self, element):

if self.\_\_length == 0:

self.\_\_first\_\_ = Node(element)

self.\_\_last\_\_ = self.\_\_first\_\_

else:

self.\_\_last\_\_.\_\_next\_\_ = Node(element, prev=self.\_\_last\_\_)

self.\_\_last\_\_ = self.\_\_last\_\_.\_\_next\_\_

self.\_\_length += 1

def clear(self):

self.\_\_length = 0

self.\_\_first\_\_ = None

self.\_\_last\_\_ = None

def pop(self):

if self.\_\_length == 0:

raise IndexError("LinkedList is empty!")

else:

if self.\_\_last\_\_.\_\_prev\_\_ == None:

self.\_\_last\_\_ = None

self.\_\_first\_\_ = None

else:

self.\_\_last\_\_ = self.\_\_last\_\_.\_\_prev\_\_

self.\_\_last\_\_.\_\_next\_\_ = None

self.\_\_length -= 1

def popitem(self, element):

current = None

is\_exists = False

for node in self:

if node.get\_data() == element:

is\_exists = True

current = node

if is\_exists:

if current is self.\_\_last\_\_:

if self.\_\_last\_\_.\_\_prev\_\_ == None:

self.\_\_last\_\_ = None

self.\_\_first\_\_ = None

else:

self.\_\_last\_\_ = self.\_\_last\_\_.\_\_prev\_\_

self.\_\_last\_\_.\_\_next\_\_ = None

elif current is self.\_\_first\_\_:

if self.\_\_last\_\_.\_\_prev\_\_ == None:

self.\_\_last\_\_ = None

self.\_\_first\_\_ = None

else:

self.\_\_first\_\_ = self.\_\_first\_\_.\_\_next\_\_

self.\_\_first\_\_.\_\_prev\_\_ = None

else:

current.\_\_next\_\_.\_\_prev\_\_ = current.\_\_prev\_\_

current.\_\_prev\_\_.\_\_next\_\_ = current.\_\_next\_\_

self.\_\_length -= 1

else:

raise KeyError("{} doesn't exist!".format(element))